OECD Healthcare data explorer

Summer semester, 2024

University of St. Gallen

Skills: Programming with Advanced Computer Languages

Authors:	**Student ID:**	**Nickname on Coding@HSG:**
Jonas Aufdenblatten		KingKong99
Adrian Furrer		
Lucas Gericke	18-610-253	LG
Oskar von Reichenbach	19-609-676	computerman
Milosz Stoj	22-601-025	Szolim

Contents:

1. Description and Use case

a. General Description

The basic premise of this program, called "Healthcare Explorer," is to compile and present data regarding healthcare across various levels of granularity for a selected number of countries, most of which are members of the OECD Organization. The data is sourced from the OECD data website and covers the years 2015 up to the present day. The software can be updated with newer data once it's released in the future, allowing for long-term usability. More details on specific data sets available will be discussed in the following subsections.

b. User input and program output

The program operates a graphical interface that allows the user to interact with it through sliders and dropdown menus, changing the scope and mode of presentation of the data. User input includes:

- Country
- Starting and ending years for the data presented
- Healthcare topic for the chart (see below)
- Type of chart (e.g., line, scatter, bar, etc.)

The program's output consists of basic information about the selected country (e.g., shape, GDP, population, currency, and language) and charts depicting the selected healthcare topics mentioned above. These healthcare topics have been chosen to reflect potential users' interests and needs regarding the markets. The envisioned users is an investors active in the pharmaceutical area (more on that in the "Use Case" section). Below the list of available topics:

- Expenditure in healthcare across areas of care over the years
- Total expenditure in healthcare across countries - peer countries
- Expenditure for prescription (Rx) and over-the-counter (OTC) over the years
- Expenditure for prescription (Rx) and over-the-counter (OTC) last year
- Absolute per capita values of Rx and OTC pharmaceutical sales over the years
- Volumes penetration (in %) of generic drugs (Gx) over the years
- Sales penetration (in %) of generic drugs (Gx) over the years
- Absolute per capita values of Gx and originator pharmaceutical sales over the years
- Number of doctors active in a country by region

c. Use case

The idea for this project comes from the real-life experience of one of the team members who was involved in assisting an investor to understand several pharmaceutical markets in Europe. Using the available OECD website to explore the data proved to be inefficient due to extremely long loading times, and the need to follow multiple data pathways to access the desired information.

Our program addresses this issue by allowing a user to access all important macro-level data about a country and its healthcare system from a single platform, The datasets chosen for our program are selected specifically due to their importance for investors:

- Share of Spend on Pharmaceuticals: This can show the importance of this segment for decision-makers and their decisiveness in potential price negotiations.
- Number of Doctors: This can indicate the number of needed sales force per region to promote a drug.
- Penetration of Generic or Originator Drugs in Price and Volume: This not only shows which segment has better prospects but also the price trends in the industry.

- OTC (Over-the-Counter) and Rx (Prescription) Split: This can inform the investor about the importance of different market segments.

Therefore, our program presents a high value-added potential for the specified user group.

2. Code: main

a. Import necessary libraries

Ensure to install all libraries via requirements.txt using pip install -r requirements.txt

```
customtkinter
Pillow
pandas
numpy
matplotlib
requests
openpyxl
```

b. Define styles and colors

Customize the appearance and theme of the application, as well as some colors

```
define styles and colors
ctk.set_appearance_mode("system") # sets mode to either light, dark or system
settings
ctk.set_default_color_theme("green") # sets default color theme to green, from
green, blue and dark
main_color = "#137C4C"
secondary_color = "#1ABE73"
widget_bg_dark = "#5D5D5D"
widget_bg_light = "#C2C2C2"
white = "#FFFFFF"
black = "#000000"
light_grey = "#ABB0B5"
dark_grey = "#4B4D50"
```

c. Check and download necessary data

Verify if necessary data files exist. If not, download them showing a progress bar.

```
file_paths = ["Data/Doctors_Region.csv", "Data/HC_Market.csv",
"Data/HC_Market_meaning.csv"]

# check if all files exist
if not all(os.path.exists(file_path) for file_path in file_paths):

      class Root(ctk.CTk): # create window for progress bar
            def __init__(self):
                  super().__init__()

                  # set title and geometry of root
                  self.title("App Name")
                  self.geometry("400x200")

                  # create descriptive label
                  self.csv_download_label = ctk.CTkLabel(self, text="Downloading
                  data...", font=("Helvetica
                  self.csv_download_label.pack(pady=30)

                  # create progress bar
                  self.csv_pb = ctk.CTkProgressBar(self, width=250,
                  progress_color=main_color)
                  self.csv_pb.pack(pady=10)
```

```
                        self.csv_pb.set(0)


                        # create progress label
                        self.csv_progress_label = ctk.CTkLabel(self, text="",
                        font=("Helvetica", 12))
                        self.csv_progress_label.pack(pady=30)
                        # start the download process
                        self.after(100, self.start_download) # add some time for gui to
                        initialize
                  def start_download(self): # set up labels and progress bar to start
                  importing
                        self.csv_progress_label.configure(text="Importing module...")
                        self.csv_pb.set(0)
                        self.after(100, self.import_module) # continue with next
                        function

                  def import_module(self): # import the necessary functions to load
                  data
                        global load_basic_oecd_data, load_oecd_med_data
                        module = importlib.import_module('total_data')
                        load_basic_oecd_data = module.load_basic_oecd_data
                        load_oecd_med_data = module.load_oecd_med_data
                        self.csv_progress_label.configure(text="Loading basic OECD
                        data...")
                        self.csv_pb.set(0.33)
                        self.after(500, self.load_basic_data) # Wait 0.5 second before
                        next step

                  def load_basic_data(self): # load in basic oecd data
                        self.csv_progress_label.configure(text="Loading med OECD
                        data...")
                        load_basic_oecd_data()
                        self.csv_pb.set(0.66)
                        self.after(1000, self.load_med_data) # Wait 1 second before
                        next step

                  def load_med_data(self): # load in med data
                        load_oecd_med_data()
                        self.csv_pb.set(1.0) # set progress to 100% when finished
                        self.csv_download_label.configure(text="Download finished. Main
                        app initializing..."
                        font=("Helvetica Bold", 16))
                        self.csv_progress_label.configure(text="Loading data
                        finished.")
                        self.after(3000,
                        self.close_window) # Wait 5 second before closing to give time
                        to read further

                  def close_window(self):
                        self.destroy()
                        print("All necessary files created.")

            root = Root()
            root.mainloop()

else:
      print("All necessary files in directory.")
```

d. **Load data into a dataframe**

Load data into a DataFrame.

```
# create dataframe from .csv files
master_df = pd.read_csv("Data/HC_Market.csv").set_index("REF_AREA")
```

e. Define custom widgets

Create create classes for the general information display and the sliders in the filter options. It also includes the MessageWindow class to create top level windows for error messages whose code is not displayed here.

```
# initialize SubFrame class for general information display
class SubFrame(ctk.CTkFrame):
	def __init__(self, master, text="", text2="-", **kwargs):

		# set default values for geometry and layout
		super().__init__(master, width=170, height=80, corner_radius=15,
		fg_color=secondary_color,

		# create label to label data displayed
		self.gi_datalabel = ctk.CTkLabel(self, text=text, font=("Helvetica
		Bold", 16), fg_color=
		self.gi_datalabel.place(x=85, y=10, anchor="n")
		self.pack_propagate(False) # stop the frame from resizing to fit the
		label

		# create label to display the data
		self.gi_data = ctk.CTkLabel(self, text=text2, font=("Helvetica", 14),
		fg_color="transparent"
		self.gi_data.place(x=85, y=40, anchor="n")

	# define function to change displayed data in gi_data
	def change_gi_data(self, new_data):
		self.text2 = new_data
		self.gi_data.configure(text=self.text2)

# initialize frame class for cool sliders with labels
class Sliders(ctk.CTkFrame):
	def __init__(self, master, steps, width, start, end, text, color_right,
	color_left):
		super().__init__(master, fg_color="transparent")

		# Create a slider
		self.slider = ctk.CTkSlider(self, width=width, number_of_steps=steps,
		from_=start, to=end, button_hover_color=main_color,
		fg_color=color_right, progress_color=color_command=self.update_label)
		self.slider.pack() #

		# Create a label to display the slider value
		self.slider_value_label = ctk.CTkLabel(self, text=text,
		font=("Helvetica", 12))
		self.slider_value_label.pack()

	# define function to update label text according to slider position
	def update_label(self, value):
		self.slider_value_label.configure(text=int(value))
```

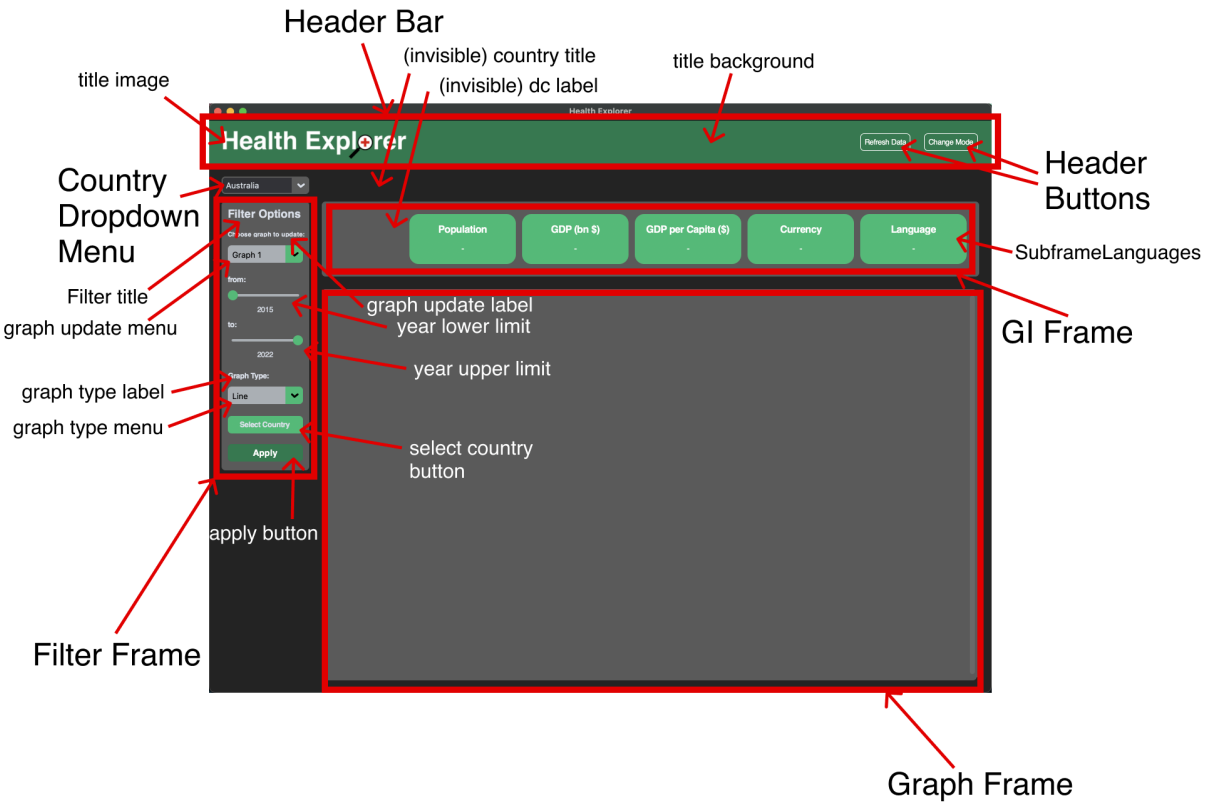
f. Define main application window

Create the main application window. It contains all widgets as well as all functions needed to display information on the front end. The only exception is the class for the “select country button” in the filter menu. It is in a separate file called countryselect.py.

It is structured the following way:

First, the class is created and some general settings are defined. Then, all the permanently visible widgets are created, in the following order. The order loosely follows a top to bottom, left to right approach.

- Header bar and title
- Header buttons
- Country dropdown menu
- Filter frame including all widgets in it from top to bottom
- General information frame including country title and subframes
- Graph frame including the grid used by the plots



Then, error message windows are initialized from the MessageWindow class, to be filled with the respective functions.

Next, the display_country_data function follows. It executes all necessary display functions when a country is chosen.

```
3 usages
354     def display_country_data(self, event=None):  # main function that runs all other functions
355         self.display_general_information()
356         self.display_graph_1()
357         self.display_graph_2()
358         self.display_graph_3()
359         self.display_graph_4()
360         self.display_graph_5()
361         self.display_graph_6()
362         self.display_graph_7()
363         self.display_graph_8()
```

Then, the display_general_information function fills the GI frame with data. It is based on the input from the country dropdown menu and creates a dynamic title, image display and general data display in the subframes.

Then, eight display_graph_x functions follow. They mainly execute the display_data_x function, set the necessary inputs as well as defaults and deal with certain errors.

They all follow this structure:

- Check for line graph error (cannot display a line graph for just one value on x-axis)
- Convert country dropdown input to country iso code
- Pass different colors to the plotting functions depending on light/dark mode
- Execute the function
- Create and place a title to display above the plot
- Destroy the previous graph if it existed
- Create and place a canvas widget to display the plot
- Deal with ValueError by displaying an error message
- Deal with AttributeError by displaying an error message

Then, the open_country_selection function creates the window for the checkboxes when you click the “Select Countries” button.

The apply_filter function runs the display_graph_x function with the parameters from the filter frame.

The update_upper_limit binds the slider (to) to the slider (from), so that there is never a negative time space (“from year” is always smaller or equal to “to year”).

The refresh_data function is repeated to download the data and display the loading bar when the refresh data button is pressed.

The change_mode function changes the mode from light to dark and dark to light when the change mode button is pressed.

Then, five error messages follow:

- open_line_graph_error: Handle the case that there is just one point on the x-axis and no line graph can be displayed.
- open_graph_type_error: Handle the case that there is a graph type (“line”, “scatter”, “bar”) is chosen that cannot be displayed in the plot
- open_no_countries_selected_error: Some graphs compare countries directly. They require some input. Usually, that is done directly by choosing a country in the country dropdown menu. However, sometimes when updating the graph via the filter frame and there are no countries selected by the checkboxes, there might be the case that no data is displayed because no countries are chosen.

- open_pop_file_does_not_exist_error: When a function requires the population_filtered.csv to execute properly and it is not present in the Data directory, this error will pop up.
- open_no_data_filtered_error: Shows on AttributeError of the display_graph_x functions. Sometimes when the filter settings are set in a way that there is no data to be displayed, there is an Attribute Error that is dealt with with this message.

Lastly, the whole class is initialized as ‘app’ and the main loop is run.

```
1065    # execute customtkinter
1066    app = App()
1067    app.mainloop()
```

6. Code: oecd_countries

a. Listing all OECD countries which can be selected in the app

```
# list with oecd countries
            oecd_countries = [
            "Australia", "Austria", "Belgium", "Canada", "Chile", "Colombia",
            "Costa Rica", "Denmark", "Estonia", "Finland",
            "France", "Germany", "Greece", "Hungary", "Iceland", "Ireland",
            "Israel", "Italy", "Japan", "Korea", "Latvia", "Lithuania",
            "Luxembourg", "Mexico", "Netherlands", "Norway",
            "Poland", "Portugal", "Slovak Republic", "Slovenia", "Spain",
            "Sweden", "Switzerland", "United Kingdom", "United States"
            ]
```

b. Listing countries which can be selected for graph 2 (due to data availability)

```
# list with countries for the select multiple countries (graph 2) feature
countries_for_country_select = [
            "Switzerland", "Australia", "Canada", "Chile",
            "Costa Rica", "Finland",
            "Poland", "United Kingdom"
            ]
```

c. Dictionary needed to assign country image in the main app window

```
# dictionary that assigns an image to each country
            oecd_images = {
            "australia": "Images/Australia.png",
            "austria": "Images/Europe/Austria.png",
            "belgium": "Images/Europe/Belgium.png",
            "canada": "Images/Canada.png",
            "chile": "Images/Chile.png",
            "colombia": "Images/Colombia.png",
            "costa rica": "Images/Costa_Rica.png",
            "denmark": "Images/Europe/Denmark.png",
            "estonia": "Images/Europe/Estonia.png",
            "finland": "Images/Europe/Finland.png",
            "france": "Images/Europe/France.png",
            "germany": "Images/Europe/Germany.png",
            "greece": "Images/Europe/Greece.png",
            "hungary": "Images/Europe/Hungary.png",
            "iceland": "Images/Europe/Iceland.png",
            "ireland": "Images/Europe/Ireland.png",
            "israel": "Images/Israel.png",
            "italy": "Images/Europe/Italy.png",
            "japan": "Images/Japan.png",
            "korea": "Images/Korea.png",
            "latvia": "Images/Europe/Latvia.png",
            "lithuania": "Images/Europe/Lithuania.png",
            "luxembourg": "Images/Europe/Luxembourg.png",
            "mexico": "Images/Mexico.png",
            "netherlands": "Images/Europe/Netherlands.png",
            "norway": "Images/Europe/Norway.png",
            "poland": "Images/Europe/Poland.png",
            "portugal": "Images/Europe/Portugal.png",
            "slovak republic": "Images/Europe/Slovakia.png",
            "slovenia": "Images/Europe/Slovenia.png",
            "spain": "Images/Europe/Spain.png",
            "sweden": "Images/Europe/Sweden.png",
            "switzerland": "Images/Europe/Switzerland.png",
```

```
            "united kingdom": "Images/Europe/United_Kingdom.png",
            "united states": "Images/United_States.png"
            }
```

d. Dictionary of ISO codes for each country used in functions across the program

```
# Dictionary of OECD countries and their ISO country codes
            oecd_iso_access = {
            "Australia": "AUS",
            "Belgium": "BEL",
            "Chile": "CHL",
            "Costa Rica": "CRI",
            "Denmark": "DNK",
            "Germany": "DEU",
            "Estonia": "EST",
            "Finland": "FIN",
            "France": "FRA",
            "Greece": "GRC",
            "Ireland": "IRL",
            "Iceland": "ISL",
            "Israel": "ISR",
            "Italy": "ITA",
            "Japan": "JPN",
            "Canada": "CAN",
            "Colombia": "COL",
            "Korea": "KOR",
            "Latvia": "LVA",
            "Lithuania": "LTU",
            "Luxembourg": "LUX",
            "Mexico": "MEX",
            "New Zealand": "NZL",
            "Netherlands": "NLD",
            "Norway": "NOR",
            "Austria": "AUT",
            "Poland": "POL",
            "Portugal": "PRT",
            "Sweden": "SWE",
            "Switzerland": "CHE",
            "Slovak Republic": "SVK",
            "Slovenia": "SVN",
            "Spain": "ESP",
            "Hungary": "HUN",
            "United Kingdom": "GBR",
            "United States": "USA"
            }
```

e. List of all graphs which include multiple countries, needed as an input in one of the functions in “main.py”.

```
# all graphs that compare multiple countries
            cross_country_graphs = ["Graph 2"]
```

7. Code: countryselect

a. Importing libraries and defining a variable to storing selected countries

```
import customtkinter as ctk

# all countries selected with checkmarks
selected_countries = ["Switzerland"]
```

b. Defining the Country Select class

This section defines a class CountrySelect that inherits from ctk.CTkToplevel. The __init__ method initializes an instance of CountrySelect. It sets the window title to "Select Country". It takes two optional parameters: countries (a list of countries) and selected_countries (a list of selected countries

```
class CountrySelect(ctk.CTkToplevel):
      def __init__(self, master=None, countries=[], selected_countries=[]):
            super().__init__(master)
            self.title("Select Country")
            self.countries = countries
            self.selected_countries = selected_countries
```

c. Check buttons

This section creates check buttons for each country based on the provided list of countries and the list of selected countries. It initializes self.check_buttons as an empty list to store the created check buttons. It iterates over the countries list and creates a check button for each country. The on or off value for each check button's variable is determined based on whether the country is in the selected_countries list. Check buttons are arranged in a grid layout with 3 columns per row. The created check buttons are appended to the self.check_buttons list.Creating check button for each country

```
            # Create check buttons for each country
            self.check_buttons = []
            self.index_var = 0
            for i, country in enumerate(self.countries):
                  var_value = "on" if country in self.selected_countries else
                  "off"
                  self.var = ctk.StringVar(value=var_value)
                  self.button = ctk.CTkCheckBox(self, text=country,
                  variable=self.var, onvalue="on", offvalue="off")
                  row = self.index_var // 3
                  column = self.index_var % 3
                  self.button.grid(row=row, column=column, padx=15, pady=10,
                  sticky="nw")
                  self.check_buttons.append(self.button)
                  self.index_var += 1

            print(self.check_buttons)
```

d. Confirm button

This section adds a "Confirm" button to the window. The button is an instance of ctk.CTkButton with the text "Confirm" and a command linked to the confirm method. The button is placed below the check buttons with appropriate padding and alignment.

```
            # Add a confirm button
            self.confirm_button = ctk.CTkButton(self, text="Confirm",
            command=self.confirm)
```

```
        self.confirm_button.grid(row=row + 1, column=1, padx=10, pady=20,
        sticky="nsew")
```

e. Confirm function

This section defines the confirm method, which is called when the "Confirm" button is clicked. It clears the selected_countries list to remove any previously selected countries. It iterates over the check buttons and adds the text of each checked button to the selected_countries list. The method prints the updated selected_countries list and closes the window using self.destroy(). This documentation provides an overview of each part of the code and explains its purpose and functionality.

```
    def confirm(self):
    self.selected_countries.clear()
    for button in self.check_buttons:
        if button.cget("variable").get() == "on":
        country_name = button.cget("text")
        self.selected_countries.append(country_name)
    print(self.selected_countries)
    self.destroy()
```

8. Code: display_data_functions

a. Importing Libraries and Helper Functions:

This section imports necessary libraries such as Matplotlib, Pandas, NumPy, and textwrap. It also defines helper functions like wrap_text for wrapping long text, and currency_formatter for formatting currency.

```
from matplotlib.figure import Figure

import matplotlib.pyplot as plt

import pandas as pd

from matplotlib.ticker import MaxNLocator, FuncFormatter, PercentFormatter

import textwrap

import numpy as np



def wrap_text(text, width): # wrap text of long legends
    wrapped_lines = textwrap.wrap(text, width=width, break_long_words=False,
    replace_whitespace=False)
    return '\n'.join(wrapped_lines)



def currency_formatter(x, pos): # format y-axis to display bn $

    return f'${x*1e-9:.1f}bn'
```

b. Add New Column If Not Exists Function

This function checks if a specific column exists in the DataFrame and adds a new column (PT_SL_VAL_TOTAL) if it doesn't exist.

```
def add_new_column_if_not_exists(data): # add new data columns
	new_column_name = 'PT_SL_VAL_TOTAL'
	if new_column_name not in data.columns:
		if 'PT_SL_VAL_M' in data.columns and 'TOTAL_PHARM' in data.columns:
			data[new_column_name] = data['PT_SL_VAL_M'] *
		data['TOTAL_PHARM']
		else:
			raise ValueError("Required columns 'PT_SL_VAL_M' and
			'TOTAL_PHARM' are missing in the data")
	return data
```

c. Add Meaning to CSV Function:

This function adds meaning to a CSV file by appending a new row with the column name and its meaning

```
def add_meaning_to_csv(csv_path, column_name, meaning):
	# Read the existing CSV file
	df = pd.read_csv(csv_path)

	# Check if the column name already exists in the DataFrame
	if column_name not in df.columns:
		# Append a new row with the column name and its meaning
		new_row = pd.DataFrame([[column_name, meaning]], columns=["Column
		name", "Meaning"])
		df = pd.concat([df, new_row], ignore_index=True)
		# Write the updated DataFrame back to the CSV file
		df.to_csv(csv_path, index=False)
		# display data for graph 1
```

d. Display Data 1 Function

This function displays data for graph 1, which represents health care cost over time. It filters data by country and time period, creates legends, and plots the data accordingly

```
def display_data_1(facecolor, labelcolor, data, plottype, selectedcountry,
selectedtimestart=2015, selectedtimeend=2023
	# filter data
	data = data[data.index.values == selectedcountry] # filter country
	data = data[(data.TIME_PERIOD <= selectedtimeend) & (data.TIME_PERIOD >=
	selectedtimestart)] # filter years

	# remove the doubled columns
	del data["HC511_y"]
	del data["HC512_y"]
	data.rename(columns={"HC511_x": "HC511", "HC512_x": "HC512"}, inplace=True)

	# import legend
	legendnames = pd.read_csv("Data/HC_Market_meaning.csv")

	# Create a dictionary mapping categories to their explanations
```

```
            legend_mapping = dict(zip(legendnames["Column name"],
            legendnames["Meaning"]))

            # Create plot
            fig = Figure(figsize=(8, 4), facecolor=facecolor)
            ax = fig.add_subplot()

            colors = ["#137C4C", "#1E88E5", "#D32F2F", "#8E24AA", "#FDD835", "#43A047",
            "#FB8C00", "#3949AB"]

            categories = ["HC0", "HC1HC2", "HC511", "HC3", "HC4", "HC512", "HC513",
            "HC52"]

            if data[categories].isnull().all().all(): # show no data available if there
            is no data available
                  fig, ax = plt.subplots()
                  ax.text(0.5, 0.5, 'No data available', fontsize=20, ha='center',
                  va='center')
                  ax.set_axis_off()

            else:
                  for i, category in enumerate(categories):
                        if plottype == "line":
                              ax.plot(data["TIME_PERIOD"], data[category],
                              label=wrap_text(legend_mapping[category], 20),
                              color=colors[i]
                        elif plottype == "scatter":
                              ax.scatter(data["TIME_PERIOD"], data[category],
                              label=wrap_text(legend_mapping[category], 20),
                              color=colors[i]
                        elif plottype == "bar":
                              ax.bar(data["TIME_PERIOD"] + i * 0.1, data[category],
                              label=wrap_text(legend_mapping[category], 20), width=
                        else:
                              raise ValueError("Unsupported plot type")

                  # Adjust the layout to make room for the legend
                  fig.subplots_adjust(right=0.75)

                  # Position the legend outside the plot area with Helvetica font
                  legend = ax.legend(title="Categories", loc='center left',
                  bbox_to_anchor=(1, 0.5), prop={'family': 'Helvetica'})
                  legend.get_title().set_fontsize('12') # set the title font size
                  legend.get_title().set_fontname('Helvetica') # set the title font
                  family


            # Set x-axis to only show integers
            ax.xaxis.set_major_locator(MaxNLocator(integer=True))

            ax.tick_params(labelsize=10, labelcolor=labelcolor,
            labelfontfamily="Helvetica")
```

```
        ax.set_title("Health Care Cost Over Time", color=labelcolor,
        fontname="Helvetica")
        ax.set_xlabel("Year", color=labelcolor, fontname="Helvetica")
        ax.set_ylabel("Health Care Cost", color=labelcolor, fontname="Helvetica")

        # Apply custom formatter to y-axis to display bn
        ax.yaxis.set_major_formatter(FuncFormatter(currency_formatter))



        return fig
```

e. Display Data 2 Function

This function displays data for graph 2, which represents total health care expenditure over time for selected countries. It filters data, creates legends, and plots the data accordingly.

```
# display data for graph 2

def display_data_2(facecolor, labelcolor, data, plottype, selectedcountry,
othercountries, selectedtimestart=2015, selectedtimeend=
        # filter data
        othercountries = np.append(othercountries, selectedcountry) # get list of
        all countries
        data = data[np.isin(data.index.values, othercountries)] # filter the
        specified countries
        data = data[(data.TIME_PERIOD <= selectedtimeend) & (data.TIME_PERIOD >=
        selectedtimestart)] # filter years

        # remove the doubled columns
        del data["HC511_y"]
        del data["HC512_y"]
        data.rename(columns={"HC511_x": "HC511", "HC512_x": "HC512"}, inplace=True)

        # create column for sum of all values
        data['TOTAL_HC_EXP'] = data.iloc[:, 1:10].sum(axis=1)
        data = data.dropna(subset=data.columns[:10])
        data = data.reset_index()

        # Create plot
        fig = Figure(figsize=(8, 4), facecolor=facecolor)
        ax = fig.add_subplot()
        colors = ["#137C4C", "#1E88E5", "#D32F2F", "#8E24AA", "#FDD835", "#43A047",
        "#FB8C00", "#3949AB"]
        countries = data.REF_AREA.values

        countries = np.unique(countries)



if data.empty:
        fig, ax = plt.subplots()
        ax.text(0.5, 0.5, 'No data available', fontsize=20, ha='center',
        va='center')
```

```
            ax.set_axis_off()

        else:
            for i, country in enumerate(countries):
                if plottype == "line":
                    ax.plot(data[data.REF_AREA == country]["TIME_PERIOD"],
                    data[data.REF_AREA == country]["TOTAL_HC_EXP"], label=country,
                    color=colors[i])
                elif plottype == "scatter":
                    ax.scatter(data[data.REF_AREA == country]["TIME_PERIOD"],
                    data[data.REF_AREA == country]["TOTAL_HC_EXP"], label=country,
                    color=colors[i])
                elif plottype == "bar":
                    ax.bar(data[data.REF_AREA == country]["TIME_PERIOD"] + i * 0.1,
                    data[data.REF_AREA == country]["TOTAL_HC_EXP"], label=country,
                    color=colors[i])
                else:
                    raise ValueError("Unsupported plot type")

            # Set x-axis to only show integers
            ax.xaxis.set_major_locator(MaxNLocator(integer=True))
            ax.tick_params(labelsize=10, labelcolor=labelcolor)
            ax.set_title("Total Health Care Expenditure Over Time", color=labelcolor)
            ax.set_xlabel("Year", color=labelcolor)
            ax.set_ylabel("Health Care Expenditure", color=labelcolor)

            # Apply custom formatter to y-axis to display bn
            ax.yaxis.set_major_formatter(FuncFormatter(currency_formatter))

            # Adjust the layout to make room for the legend
            fig.subplots_adjust(right=0.75)

            # Position the legend outside the plot area with Helvetica font
            legend = ax.legend(title="Countries", loc='center left', bbox_to_anchor=(1,
            0.5))
            legend.get_title().set_fontsize('12') # Optional: set the title font size

            return fig
```

f. **Display Data 3 Function:**

This function displays data for graph 3, which represents expenditure on prescription vs. over-the-counter drugs over time. It filters data, creates legends, and plots the data accordingly.

```
        # display data for graph 3
            def display_data_3(facecolor, labelcolor, data, plottype, selectedcountry,
            selectedtimestart=2015, selectedtimeend=2023

            # filter data
            data = data[data.index.values == selectedcountry] # filter country
            data = data[(data.TIME_PERIOD <= selectedtimeend) & (data.TIME_PERIOD >=
            selectedtimestart)] # filter years

            # remove the doubled columns
```

```
            data.rename(columns={"HC511_x": "HC511", "HC512_x": "HC512"}, inplace=True)

            # import legend
            legendnames = pd.read_csv("Data/HC_Market_meaning.csv")

            # Create a dictionary mapping categories to their explanations
            legend_mapping = dict(zip(legendnames["Column name"],
            legendnames["Meaning"]))

            # Create plot
            fig = Figure(figsize=(8, 4), facecolor=facecolor)
            ax = fig.add_subplot()

            colors = ["#137C4C", "#1E88E5", "#D32F2F", "#8E24AA", "#FDD835", "#43A047",
            "#FB8C00", "#3949AB"]


            categories = ["HC511", "HC512"]


            if data[categories].isnull().all().all(): # show no data available if there
            is no data available
                  fig, ax = plt.subplots()
                  ax.text(0.5, 0.5, 'No data available', fontsize=20, ha='center',
                  va='center')
                  ax.set_axis_off()
            else:
                  for i, category in enumerate(categories):
                        if plottype == "line":
                              ax.plot(data["TIME_PERIOD"], data[category],
                              label=wrap_text(legend_mapping[category], 20),
                              color=colors[i])
                        elif plottype == "scatter":
                               label=wrap_text(legend_mapping[category], 20),
                              color=colors[i])
                        elif plottype == "bar":
                              ax.bar(data["TIME_PERIOD"] + i * 0.1, data[category],
                              label=wrap_text(legend_mapping[category], 20),
                              width=0.1, color=colors[i])
                        else:
                              raise ValueError("Unsupported plot type")

                  # Adjust the layout to make room for the legend
                  fig.subplots_adjust(right=0.75)

                  # Position the legend outside the plot area with Helvetica font
                  legend = ax.legend(title="Categories", loc='center left',
                  bbox_to_anchor=(1, 0.5), prop={'family': 'Helvetica'})
                  legend.get_title().set_fontsize('12') # set the title font size
                  legend.get_title().set_fontname('Helvetica') # set the title font
                  family

                  # Set x-axis to only show integers
```

```
        ax.xaxis.set_major_locator(MaxNLocator(integer=True))
        ax.tick_params(labelsize=10, labelcolor=labelcolor,
        labelfontfamily="Helvetica")
        ax.set_title("Expenditure Prescription vs OTC Over Time",
        color=labelcolor, fontname="Helvetica")
        ax.set_xlabel("Year", color=labelcolor, fontname="Helvetica")
        ax.set_ylabel("Expenditure", color=labelcolor, fontname="Helvetica")

        # Apply custom formatter to y-axis to display bn
        ax.yaxis.set_major_formatter(FuncFormatter(currency_formatter))

        return fig
```

g. Display Data 4 Function

This function displays data for graph 4, which represents expenditure on prescription vs. over-the-counter drugs per capita over time. It filters data, calculates spend per capita, creates legends, and plots the data accordingly.

```
# display data for graph 4

def display_data_4(facecolor, labelcolor, data, plottype, selectedcountry,
selectedtimestart=2015, selectedtimeend=2023


    # filter data
    data = data[data.index.values == selectedcountry] # filter country
    data = data[(data.TIME_PERIOD <= selectedtimeend) & (data.TIME_PERIOD >=
    selectedtimestart)] # filter years

    # remove the doubled columns
    data.rename(columns={"HC511_x": "HC511", "HC512_x": "HC512"}, inplace=True)

    # Read population data
    population_data = pd.read_csv("Data/population_filtered.csv")

    # Merge population data with your existing DataFrame
    data = pd.merge(data, population_data, on=["REF_AREA", "TIME_PERIOD"],
    how="left")

    # Calculate spend per capita
    data["HC511"] = data["HC511"] / data["Total_Population"]
    data["HC512"] = data["HC512"] / data["Total_Population"]

    # import legend
    legendnames = pd.read_csv("Data/HC_Market_meaning.csv")

    # Create a dictionary mapping categories to their explanations
    legend_mapping = dict(zip(legendnames["Column name"],
    legendnames["Meaning"]))

    # Create plot
    fig = Figure(figsize=(8, 4), facecolor=facecolor)
```

```
            ax = fig.add_subplot()

            colors = ["#137C4C", "#1E88E5", "#D32F2F", "#8E24AA", "#FDD835", "#43A047",
            "#FB8C00", "#3949AB"]



            categories = ["HC511", "HC512"]


            if data[categories].isnull().all().all(): # show no data available if there
            is no data available
                  fig, ax = plt.subplots()
                  ax.text(0.5, 0.5, 'No data available', fontsize=20, ha='center',
                  va='center')
                  ax.set_axis_off()

            else:
                  for i, category in enumerate(categories):
                        if plottype == "line":
                              ax.plot(data["TIME_PERIOD"], data[category],
                              label=wrap_text(legend_mapping[category], 20),
                              color=colors[i])
                        elif plottype == "scatter":
                              ax.scatter(data["TIME_PERIOD"], data[category],
                              label=wrap_text(legend_mapping[category], 20),
                              color=colors[i])
                        elif plottype == "bar":
                              ax.bar(data["TIME_PERIOD"] + i * 0.1, data[category],
                              label=wrap_text(legend_mapping[category], 20),
                              width=0.1, color=colors[i])
                        else:
                              raise ValueError("Unsupported plot type")

                  # Adjust the layout to make room for the legend
                  fig.subplots_adjust(right=0.75)

                  # Position the legend outside the plot area with Helvetica font
                  legend = ax.legend(title="Categories", loc='center left',
                  bbox_to_anchor=(1, 0.5), prop={'family': 'Helvetica'})
                  legend.get_title().set_fontsize('12') # set the title font size
                  legend.get_title().set_fontname('Helvetica') # set the title font
                  family

            # Set x-axis to only show integers
            ax.xaxis.set_major_locator(MaxNLocator(integer=True))
            ax.tick_params(labelsize=10, labelcolor=labelcolor,
            labelfontfamily="Helvetica")
            ax.set_title("Expenditure Prescription vs OTC Over Time", color=labelcolor,
            fontname="Helvetica")
            ax.set_xlabel("Year", color=labelcolor, fontname="Helvetica")
            ax.set_ylabel("Expenditure per capita (in $)", color=labelcolor,
            fontname="Helvetica")
```

```
        return fig
```

h. **Display Data 5 Function**

This function displays data for graph 5, which represents generic penetration in volume over time. It filters data, creates legends, and plots the data accordingly.

```
# display data for graph 5

def display_data_5(facecolor, labelcolor, data, plottype, selectedcountry,
selectedtimestart=2015,

selectedtimeend=2023):


        # filter data
        data = data[data.index.values == selectedcountry] # filter country
        data = data[data.TIME_PERIOD <= selectedtimeend] # filter end year
        data = data[data.TIME_PERIOD >= selectedtimestart] # filter end year

        # import legend
        legendnames = pd.read_csv("Data/HC_Market_meaning.csv")

        # Create a dictionary mapping categories to their explanations
        legend_mapping = dict(zip(legendnames["Column name"],
        legendnames["Meaning"]))

        # Create plot
        fig = Figure(figsize=(8, 4), facecolor=facecolor)
        ax = fig.add_subplot()

        colors = ["#137C4C", "#1E88E5", "#D32F2F", "#8E24AA", "#FDD835", "#43A047",
        "#FB8C00", "#3949AB"]

        categories = ["PT_SL_VOL_M"]

        if data[categories].isnull().all().all(): # show no data available if there
        is no data available
                fig, ax = plt.subplots()
                ax.text(0.5, 0.5, 'No data available', fontsize=20, ha='center',
                va='center')
                ax.set_axis_off()

        else:
                for i, category in enumerate(categories):
                        data[category] = data[category] * 100 # Convert to percentage
                        if plottype == "line":
                                ax.plot(data["TIME_PERIOD"], data[category],
                                label=wrap_text(legend_mapping[category], 15),
                                color=colors[i])
                        elif plottype == "scatter":
```

```
                        ax.scatter(data["TIME_PERIOD"], data[category],
                        label=wrap_text(legend_mapping[category], 15),
                        color=colors[i])
                  elif plottype == "bar":
                        ax.bar(data["TIME_PERIOD"] + i * 0.1, data[category],
                        label=wrap_text(legend_mapping[category], 15),
                        width=0.5,
                        color=colors[i])
                  else:
                        raise ValueError("Unsupported plot type")

            # Adjust the layout to make room for the legend
            fig.subplots_adjust(right=0.75)

            # Position the legend outside the plot area with Helvetica font
            legend = ax.legend(title="Categories", loc='center left',
            bbox_to_anchor=(1, 0.5),
            prop={'family': 'Helvetica'})
            legend.get_title().set_fontsize('12') # set the title font size
            legend.get_title().set_fontname('Helvetica') # set the title font
            family

      # Set x-axis to only show integers
      from matplotlib.ticker import MaxNLocator
      ax.xaxis.set_major_locator(MaxNLocator(integer=True))

      # Set y-axis labels to percentages
      ax.yaxis.set_major_formatter(PercentFormatter())
      ax.tick_params(labelsize=10, labelcolor=labelcolor,
      labelfontfamily="Helvetica")
      ax.set_title("Generic penetration in volume (in %)", color=labelcolor,
      fontname="Helvetica")
      ax.set_xlabel("Year", color=labelcolor, fontname="Helvetica")
      ax.set_ylabel("Volume of pharmaceutical sales", color=labelcolor,
      fontname="Helvetica")

      return fig
```

i. Display Data 6 Function

This function displays data for graph 6, which represents generic penetration in value over time. It filters data, creates legends, and plots the data accordingly.

```
# display data for graph 6
      def display_data_6(facecolor, labelcolor, data, plottype, selectedcountry,
      selectedtimestart=2015,
      selectedtimeend=2023):

            # filter data
            data = data[data.index.values == selectedcountry] # filter country
            data = data[data.TIME_PERIOD <= selectedtimeend] # filter end year
            data = data[data.TIME_PERIOD >= selectedtimestart] # filter end year

            # import legend
```

```
                legendnames = pd.read_csv("Data/HC_Market_meaning.csv")

                # Create a dictionary mapping categories to their explanations
                legend_mapping = dict(zip(legendnames["Column name"],
                legendnames["Meaning"]))

                # Create plot
                fig = Figure(figsize=(8, 4), facecolor=facecolor)
                ax = fig.add_subplot()

                colors = ["#137C4C", "#1E88E5", "#D32F2F", "#8E24AA", "#FDD835",
                "#43A047", "#FB8C00", "#3949AB"]

                categories = ["PT_SL_VAL_M"]

                if data[categories].isnull().all().all: # show no data available if
                there is no data available
                      fig, ax = plt.subplots()
                      ax.text(0.5, 0.5, 'No data available', fontsize=20,
                      ha='center', va='center')
                      ax.set_axis_off()
                else:
                      for i, category in enumerate(categories):
                            data[category] = data[category] * 100 # Convert to
                            percentage
                            if plottype == "line":
                                  ax.plot(data["TIME_PERIOD"], data[category],
                                  label=wrap_text(legend_mapping[category], 15),
                                  color=colors[i])
                            elif plottype == "scatter":
                                  ax.scatter(data["TIME_PERIOD"], data[category],
                                  label=wrap_text(legend_mapping[category], 15),
                                  color=colors[i])
                            elif plottype == "bar":
                                  ax.bar(data["TIME_PERIOD"] + i * 0.1,
                                  data[category],
                                  label=wrap_text(legend_mapping[category], 15),
                                  width=0.5,
                                  color=colors[i])
                            else:
                            raise ValueError("Unsupported plot type")

                      # adjust layout to make room for legend
                      fig.subplots_adjust(right=0.75)

                      # Position the legend underneath the plot
                      legend = ax.legend(title="Categories", loc='center left',
                      bbox_to_anchor=(1, 0.5), prop={'family': 'Helvetica'})
                      legend.get_title().set_fontsize('12') # set the title font size
                      legend.get_title().set_fontname('Helvetica') # set the title
                      font family

                # Set x-axis to only show integers
```

```
                from matplotlib.ticker import MaxNLocator
                ax.xaxis.set_major_locator(MaxNLocator(integer=True))

                # Set y-axis labels to percentages
                ax.yaxis.set_major_formatter(PercentFormatter())
                ax.tick_params(labelsize=10, labelcolor=labelcolor,
                labelfontfamily="Helvetica")
                ax.set_title("Generic penetration in value (in %)", color=labelcolor,
                fontname="Helvetica")
                ax.set_xlabel("Year", color=labelcolor, fontname="Helvetica")
                ax.set_ylabel("Value of pharmaceutical sales", color=labelcolor,
                fontname="Helvetica")
                return fig
```

j. Display Data 7 Function

This function displays data for graph 7, which represents generic spend per capita over time. It filters data, calculates spend per capita, creates legends, and plots the data accordingly.

```
# display data for graph 7

def display_data_7(facecolor, labelcolor, data, plottype, selectedcountry="GER",
selectedtimestart=2015,

selectedtimeend=2023):


        # filter data
        data = data[data.index.values == selectedcountry] # filter country
        data = data[data.TIME_PERIOD <= selectedtimeend] # filter end year
        data = data[data.TIME_PERIOD >= selectedtimestart] # filter end year

        # Add new column if not exists
        data = add_new_column_if_not_exists(data)

        # Call the function to add meaning to the CSV file
        add_meaning_to_csv("Data/HC_Market_meaning.csv", "PT_SL_VAL_TOTAL", "Generic
        pharmaceuticals spend per capita")

        # import legend
        legendnames = pd.read_csv("Data/HC_Market_meaning.csv")

        # Read population data
        population_data = pd.read_csv("Data/population_filtered.csv")

        # Merge population data with your existing DataFrame
        data = pd.merge(data, population_data, on=["REF_AREA", "TIME_PERIOD"],
        how="left")

        # Calculate spend per capita
        data["PT_SL_VAL_TOTAL"] = data["PT_SL_VAL_TOTAL"] / data["Total_Population"]
```

```
# Create a dictionary mapping categories to their explanations
legend_mapping = dict(zip(legendnames["Column name"],
legendnames["Meaning"]))

# Create plot
fig = Figure(figsize=(8, 4), facecolor=facecolor)
ax = fig.add_subplot()

colors = ["#137C4C", "#1E88E5", "#D32F2F", "#8E24AA", "#FDD835", "#43A047",
"#FB8C00", "#3949AB"]

categories = ["PT_SL_VAL_TOTAL"]

if data[categories].isnull().all().all(): # show no data available if there
is no data available
      fig, ax = plt.subplots()
      ax.text(0.5, 0.5, 'No data available', fontsize=20, ha='center',
      va='center')
      ax.set_axis_off()

else:
      for i, category in enumerate(categories):
            data[category] = data[category]
            if plottype == "line":
                  ax.plot(data["TIME_PERIOD"], data[category],
                  label=wrap_text(legend_mapping[category], 15),
                  color=colors[i])
            elif plottype == "scatter":
                  ax.scatter(data["TIME_PERIOD"], data[category],
                  label=wrap_text(legend_mapping[category], 15),
                  color=colors[i])
            elif plottype == "bar":
                  ax.bar(data["TIME_PERIOD"] + i * 0.1, data[category],
                  label=wrap_text(legend_mapping[category], 15),
                  width=0.5,
                  color=colors[i])
            else:
            raise ValueError("Unsupported plot type")

      # Adjust the layout to make room for the legend
      fig.subplots_adjust(right=0.75)

      # Position the legend outside the plot area with Helvetica font
      legend = ax.legend(title="Categories", loc='center left',
      bbox_to_anchor=(1, 0.5),
      prop={'family': 'Helvetica'})
      legend.get_title().set_fontsize('12') # set the title font size
      legend.get_title().set_fontname('Helvetica') # set the title font
      family

# Set x-axis to only show integers
from matplotlib.ticker import MaxNLocator
ax.xaxis.set_major_locator(MaxNLocator(integer=True))
```

```
            ax.tick_params(labelsize=10, labelcolor=labelcolor,
            labelfontfamily="Helvetica")
            ax.set_title("Generic spend per capita (in dollars)", color=labelcolor,
            fontname="Helvetica")
            ax.set_xlabel("Year", color=labelcolor, fontname="Helvetica")
            ax.set_ylabel("Value of pharmaceutical sales", color=labelcolor,
            fontname="Helvetica")

            return fig
```

9. Reflection and limitations

Looking ahead, we see several opportunities to enhance our software. Improving data quality remains a priority – eliminating blank spots in the database by triangulating with other open data sources is one possible solution.

Another useful improvement would be the introduction of automatic data upload. This feature would save users time and ensure they always have the latest information without needing manual updates.

As we look ahead to future development, it's clear that investing more time in testing could significantly improve our software. Unfortunately, due to project time limitations, we couldn't allocate as much time to testing as we would have liked. However, prioritizing testing in future iterations could help catch more issues early on, making our app more reliable and user-friendly.

Additionally, further work could include adding new capabilities, such as advanced analytical tools and more customizable dashboards, to make the software more powerful and user-friendly.